Verifying a Binary Micro-Hypervisor Intercept Handler

Nicholas Carlini, Michael McCoyd, Rohit Sinha

\{npc, mmccoyd, rsinha\}@cs.berkeley.edu

Abstract

Hypervisors provide a security foundation of cloud computing, yet have suffered exploits. Efforts at formal verification have included codevelopment (XMHF) and interactive theorem proving (seL4). A technique that can be quickly applied to existing hypervisors is desirable.

We examine binaries by extending the Binary Analysis Platform (BAP) to include the required system mode instructions used by hypervisors. We translate the BAP output to Boogie, a language which is then converted to an SMT formula, and define a security property that the hypervisor response to guest execution does not modify the hypervisor code pages. We use BAP and Boogie to automatically detect one ported and two injected bugs and verify that 1000 random traces through the intercept handler do not violate our property. We identify constructs that are difficult to verify automatically. Our tool should be usable in analyzing other hypervisors.

1 Introduction

Hypervisors form the basis of cloud computing, as they are relied upon for the separation of client activities. Despite their security-critical role, and many hours of manual bugfinding, hypervisors are not without vulnerabilities. For example, a 2008 Xen exploit allowed a normal guest running a paravirtualized frame buffer to run arbitrary code in dom0, bypassing NX, ASLR and SELinux [10]. This exploit gave access to hardware resources, with the ability to read or change any disk data.

Formal verification provides one means of assurance of correctness. Our work is in the domain of formal verification using symbolic execution, focused on hypervisors. Previous work has verified hypervisors with two different methods. The XMHF hypervisor [9] was codeveloped with its proof, and seL4 [7] was shown secure with interactive theorem proving. While these efforts have produced good results applying them to different hypervisors would seem nearly as expensive as the original efforts.

It is our hope to create a tool which can automatically analyze existing hypervisor binaries for common security properties. Binary analysis allows our tool to be reusable regardless of implementation language, and without manual modeling of the program.

All security properties depend on the specifics of the hypervisor code. Thus, we chose to verify that the hypervisor code pages do not change in response to guest actions. Proving this property is necessary to prove almost any other interesting property, and the tools we develop in proving it can be applied to other properties.

With this limited property, our work does not say much about the hypervisor it analyzes. Instead, we consider the question if it is feasible to prove properties about hypervisors considering only the binary file. We believe our results provide encouragement in this area.

Contributions:

1. We develop an automated approach to verifying hypervisors using the compiled binary, which detects ported and injected bugs.
2. We run our tool on a modified NOVA hypervisor and find no bugs which fail our property.
3. We identify constructs which make verification hard, and discuss alternatives which are just as powerful but are easier to prove correct.

The remainder of the paper is laid out as follows. Section 2 provides background on hypervisors and verification. We summarize related work in Section 3, and in Section 4 we discuss the existing tools we build upon for our verification. We describe our overall approach in Section 5 and discuss the specifics of the architecture in Section 6. Section 7 discusses the security properties we verify and we run our tool on these properties in Section 8. We run our tool looking for unknown bugs in
of two ways, symbolic verification can go
to branches, which allow the flow of execution to go one
state about the formulas for each register. At conditional
computed, in symbolic verification we maintain symbolic
concrete execution
Whereas in...

Section 9. In Section 10 we discuss what difficulties we
encountered in verifying hypervisors, and discuss future
work and conclude in Sections 11 and 12.

2 Background

NOVA [8] is an academic hypervisor with a micro-
hypervisor core supporting multiple guests and multiple
CPU cores. Its protection is run by a host-mode ring-0
core (9K SLoC), while guest management and devices
are managed by host mode ring 3 modules (27K SLoC).
It is implemented in highly templated C++. We examine
just the binary of its host-mode ring-0 core.

Hypervisors are generally split into two pieces. There
is code which sets up the state of the hypervisor (the
setup code), and code which responds to intercepts gen-
generated by the guests (the intercept handler). The setup
code initializes the nested page tables, sets up hypervisor
memory, and launches the guests. The intercept handler
is responsible for dealing with guests which attempt ille-
gal behavior (e.g., by writing to a control register), deal-
ing with page-faults caused by the guest, and responding
to hypercalls which the guest has requested.

In this work, we focus on verifying the intercept han-
der for correctness. This is for two reasons. First, we
believe it to be more likely to contain bugs. While the
setup code has the luxury of running before any hostile
guest, the intercept handler must deal with the active ad-
dversary of the guest. The second reason we analyze the
intercept handler is its task is limited. In most cases the
path from the beginning of handling an intercept to re-
suming a guest is only a few hundred instructions.

We rely on symbolic verification for our proof. Wherea
concrete execution actual values are assigned
to registers and the resulting values are directly com-
puted, in symbolic verification we maintain symbolic
state about the formulas for each register. At conditional
branches, which allow the flow of execution to go one
of two ways, symbolic verification can go both ways;
concrete execution would have to split the verification
in half and continue executing both paths. Our current
trace based analysis dose not exploit symbolic verifica-
tions power, but our trace based approach is only tempo-
ary.

3 Related Work

The seL4 [4, 7] work is a large scale effort to develop
and verify a microkernel. The microkernel was specified
in Haskell and then implemented in C. The interactive
proof is 200K line of Isabella script. Development of
the kernel was estimated to take four person years and
the proof to take 20 person years. Our approach is non-
interactive and attempts to work with an existing binary,
though with a simple property.

XMHF [9] developed an extensible 6k SLoC hyper-
viser along with its proof. The CBMC model checker
was used to test for memory integrity of the C code but
manual analysis was used for assembly sections and loop
constructs. Our effort starts its analysis from the binary
code and works with a modified existing hypervisor.

MAYHEM [3] uses binary analysis to find exploits in
user mode x86 code. We do not generate exploits, but use
the BAP tool of MAYHEM with Boogie as a backend
to test for vulnerabilities in binary code. BAP (Binary
Analysis Platform) [2] is a tool which operates on x86
user mode binaries. It contains a translator from x86 to
a simple register transfer language for many user-mode
instructions. It has been used in automated vulnerability
discovery and exploit generation. We use its toil front
end, which converts x86 to its intermediate language, as
the source of our x86 model of user-mode instruction
semantics.

4 Tools

BAP Though initially designed for bugfinding and not
complete formal verification, BAP has wide coverage of
the x86 user mode semantics. We extend BAP’s Toil
front end with system mode instructions and use its in-
termediate representation to create a formal model of the
program. BAP has three general statement types that we
use. Move statements transfer a value from some location
into a variable. Store statements transfer a value into a
memory array. Jump statements redirect program flow.
Each statement may contain multiple expressions, which
can either be Loads of an array at a specific index, or
any of a number of unary or binary operations on other
expressions.

Figure 1 gives a simple x86 program which adds two
integers and stores the result at the address of register
es1 plus four. The corresponding BAP IR is shown at
right, with all flags updated as the x86 specification re-
quires. Finally the result is stored into memory.

Boogie Boogie [1] is “an Intermediate Verification
Language (IVL) for describing proof obligations to be
discharged by a reasoning engine, typically an SMT
solver.” It provides a programming language interface
that is easier to work with than raw SMT solver clauses.
Our tools produce Boogie code and we use Boogie and a
SMT solver, Z3 [6], to check satisfaction of our proper-
ties.

Boogie contains many constructs for verifying prop-
erties about structured programs, including procedures
with pre- and post-conditions, loops with invariants, and
the ability to add axioms to general functions. We do not need, or even use, most of this, instead, we just use Boogie as a simplified frontend to SMT solvers.

5 Approach Overview

Our goal in this work is to demonstrate the practicality of proving properties about hypervisors given the program binary as input. There are many difficulties we encounter when analyzing the binary program. The most difficult cases are that of indirect jumps in the assembly level and the need for loop invariants that are best extracted from the source. Our current tool verifies that individual traces of program execution satisfy a given security property. We define a trace as a single path through the binary starting from a given initial instruction and ending at any set of terminal states. Note that this current implementation can only be used for bug-finding, not for proving correctness, because we do not analyze all possible traces.

In analyzing hypervisors, a natural breakpoint in execution is the transition between guest and host execution. Setup code prepares the system for the first guest execution and the intercept handler responds to guest actions and then resumes the guest. When dealing with each, the state initially is a function of the state created by the hardware any any previous code which has executed. For a security property to exist in the face of guest execution, it must be created by the setup code and maintained by the intercept handling code. Our verification tests each of these. If we can show this, then we will be able to show our properties are always maintained.

We define and prove properties that most hypervisors intend to provide. Two examples of properties are that "Sane hypervisors prevent the guest from changing hypervisor code pages" or "Simple one cpu per guest hypervisors do not exchange information between guests". We begin with the first of these as its absence would make it very difficult to prove anything further. We can case split our property into three cases:

- Hypervisor code pages are not modified during guest execution.
- Hypervisor code pages are not modified by the hardware transition between guest and host modes.
- Hypervisor code pages are not modified by the hypervisor in response to guest execution.

For now we assume the first two cases are satisfied and start with the intercept handling code for its simplicity and its need to respond to a hostile guest.

We perform our analysis on the binary file for multiple reasons. First, is the common language hypervisors are compiled to: this allows us to verify hypervisors written in any compiled language, regardless of what the source language is. Second, it allows us to prove properties about the actual code which is executed; while unlikely, it is possible for the source code to be error free but the binary to have a bug introduced by the compiler. Third, even with all of the complexity of x86, correctly modeling the semantics of some languages would be even more difficult, for they may not even be well defined. Finally, arguing about the binary code allows us to express...
more powerful statements which may not be expressible in the original source level. For example, we may wish to prove that registers which were last written by the guest are rewritten by the host before they are read. This property could not be expressed at the C level.

Our approach is to define our property and the initial machine state in Boogie, use BAP and our BAP IR → Boogie tool to express the binary’s semantics in Boogie, and use Boogie and its SMT solver to test if our property is satisfied.

There are several implementation issues. First, Boogie jumps need explicit target addresses, so we cannot directly express call return statements to arbitrary calling addresses. Second, we are not ready to throw all of the binary at the SMT solver, so are analyzing smaller parts or traces first. Third, there are indirect jumps from linked lists of function pointers which we do not have the information to handle. An important limitation of our work is that we comment out these linked lists in the hypervisor source code. We are thus not analyzing the original hypervisor but a fatally broken one which assumes that the length of the linked list is always 0. While this is an allowable state, it is not complete. (That is, we did not introduce a change which fundamentally breaks previous code, so we only add extra assumptions about the initial state.) We hope to rectify this in future work with analysis of startup code or replacement of the linked lists with a more readily analyzed data structure. The details of our implementation are presented in Figure 2 and Section 6.

Our threat model includes a hostile guest but trusted hardware, boot loader, and a trusted but potentially buggy hypervisor. Because we start from the binary we do not rely on a trust worthy compiler at the vendor or that there are no binary changes in distribution. We do rely on our own analysis tool chain.

5.1 Analysis Execution Environment

The execution environment defined by BAP’s semantics is a simplification from the real machine. With its use in analyzing application code, BAP does not include the indirection that a page table adds to memory access. We have not added paged memory access to BAP but rather make the assumption that the hypervisor has set its own page tables as an identity map. This is sufficient for analyzing execution in host mode.

BAP defines an array of memory locations and program instructions and labels. Yet the program instructions do not exist within the memory array. Thus BAP has separate code and data address spaces instead of a single code and data address space on the x86. If we extend our analysis from “code pages are not modified” to “code pages are neither read nor modified”, and prove this, then the separate BAP address spaces should not affect the validity of our results.

6 Architecture

Our more detailed processing is shown in Figure 2 and discussed in detail in this section.

6.1 Straight-Line Binary

We begin by taking a compiled version of the hypervisor, and extracting from it a number of straight-line traces. This has the effect of simplifying our input to a binary with only one branch path. This allows us to convert the binary directly to Boogie, which requires static knowledge of jump targets, as well as reduces the solver complexity by eliminating a potentially exponential number of branches that could have been taken. This step takes the full ELF binary and transform it into several ELF binary outputs with random trace through the intercept handler.

To start out, we run objdump on the hypervisor binary to obtain the disassembly for use in the trace generation. We built a tool which extracts from the disassembled binary a series of traces by following the execution path from a given starting address to any of a given number of final addresses.

Our tool outputs a .c file with each instruction along the trace inlined. That is, when the tool encounters a jmp instruction in the assembly, it inserts a jmp into the output but changes the destination address to immediately follow that instruction.

There are three nontrivial cases when creating a trace. Call instructions can be handled similarly to jump instructions, with the added complexity that they also push a return address to the stack.

The return instructions must be rewritten. Because the call has been inlined to call the instruction directly following it, returning (back) to that instruction would not be correct. We therefore replace returns with the addition of four (on 32 bit systems) to the stack pointer.

Finally, conditional jumps must also be rewritten to note the path which was taken, so our analysis tool can place assumptions about the register state which impacted the branch choices. For example, if there is a j ne instruction which is not taken, then the analysis must assume at that point that the zero flag is not set. However, when picking random paths, simply producing the output trace would lose this information about which path was actually chosen.

To fix this, after choosing random paths, we rewrite all branch instruction so that they are not taken. The taken path (that is, the address to branch to if the branch were to succeed) jumps to the end of the program, which we define as an illegal state. If, for example, we encounter a
The next step in the process is to generate the BAP IR from the binary generated in the previous section. To do this, we extended BAP to work with system-mode binaries. This includes a model of the Virtual Machine Control Structure (VMCS), control registers, and general system-mode instructions.

The VMCS on x86 hardware is a 4k wide register which contains state about the virtual machine. It can be read with the instruction `vmread` and written to with the instruction `vmwrite`.

In order to not explode the SMT solver, we model the

\[
V_{\text{read}}(t, \text{dst}, \text{src}) \rightarrow [\text{asm } t \text{ dst}]
\]

\[
(\text{loadm } \text{vmcs } r32 (\text{op2e } t \text{ src}))
\]

\[
V_{\text{write}}(t, \text{dst}, \text{src}) \rightarrow [\text{storem } \text{vmcs } t]
\]

\[
(\text{op2e } t \text{ src}) (\text{op2e } t \text{ dst})]
\]

VMCS as a secondary memory array. This also simplifies our implementation of the `vmread/vmwrite` instructions as they become identical to `mov` instructions only with the memory referencing the VMCS instead of regular memory. The implementation of these two instructions is given in Figure 3.

We additionally add instructions to move to and from the Control Registers (CR0, CR1, CR2, and CR3), and to read the timestamp counter.

Finally, we extend BAP to keep track of the types (bit width) of all arguments for every function. This is required because the addition of two 8-bit registers has different semantics than the addition of two 16-bit registers. BAP did not keep track of this information for all functions. Instead, it only typed the variables, which is insufficient for our purposes.
6.3 Boogie Model

We construct a formal model of the BAP IR in Boogie. We do not need to model any x86 specifics because at this point every effect of the x86 instructions has been modeled in the simpler BAP IR.

Registers are modeled using 32-bit bitvectors in Boogie, when the full width of the register is used (e.g., eax is read), or 16-bit or 8-bit bitvectors if requested (e.g., ax or ah is read).

We then create a model of each of the unary and binary operations present in the BAP IR (e.g., addition, subtraction, logical and), which operate on any of 8-, 16-, and 32-bit operands. When possible, we have Boogie use the underlying SMT solver’s operations instead of reimplementing basic operations on bitvectors within Boogie.

Our model of memory can be thought of as a map from 32-bit bitvectors to 32-bit bitvectors. This allows Boogie to only model those memory addresses which have been written to, and not reason about all $2^{32}$ addresses.

The reason we choose this model instead of a map from 32-bit bitvectors to 8-bit bitvectors is performance: the majority of reads and writes are to 32-bit integers on 4-byte aligned words. This model allows us to simply load and store entire bitvectors at once.

However, this complicates the process for loading and storing values which are not 32-bits. In particular, when storing a 16-bit value we must first load the full 32-bit value from memory, overwrite the correct 16-bit part, and store the value back to memory.

Also note that while memory is technically mapping from 32-bit bitvectors to 32-bit bitvectors, in reality it is mapping from 30-bit bitvectors to 32-bit bitvectors, since the low two bits on the source are always zero.

Figure 4 has the Boogie model we constructed for storing 8-bit bitvectors into memory. It begins with the definitions of ITE (if/then/else) as picking either the second or third arguments based on the condition. Then the align function is defined to align a 32-bit value to 4-byte word boundaries. Then the actual store function is defined case by case for each of the 4 bytes that could be written to within a word. Because x86 is little-endian, an aligned write overwrites the high byte of the 32-bit word.

In a previous version of our tool, we had a bug in verifying our property which manifested itself because of our complicated memory model. The property we wrote was expressed in terms of the memory array directly instead of using the load and store functions we wrote. As such, there was a potential off-by-up-to-three error.

6.4 BAP IR to Boogie

We wrote a tool which converts the BAP IR to Boogie which can then be sent to the SMT solver (Z3 in this case). BAP and boogie have different constructions and expressiveness, and we need to convert from one to another.

While Boogie supports constructs such as loops (with invariants) and procedures (with pre- and post-conditions), we do not use these. Instead, we create only one procedure which contains the entire effects of the program. The register-transfer language in BAP maps well on to Boogie’s use of variables. Additionally, the BAP memory maps directly to an array of 32 bit integers as described above. We model all jumps within the binary as gotos within Boogie, and conditional jumps as nondeterministic gotos.

When converting a conditional jump to a nondeterministic goto, there is slightly more to do than just converting $\text{cjmp(Condition, Lab1, Lab2)}$ to $\text{goto Lab1, Lab2}$. We must observe that Condition holds when taking the first path, but does not hold when taking the second path. It would be incorrect to make the code at Lab1 begin by issuing an assume of Condition, because there may be many entry points to Lab1. We therefore convert the conditional jump to a sequence of Boogie instructions given in Figure 5. This allows the correct assumptions to be made along each code path.

We lightly optimize the Boogie output generated by our tool by removing dead-code. After each instruction, BAP’s state of the CPU is totally accurate, with all registers and flags updated correctly. However, if a flag is updated implicitly, but never read from (e.g., the parody flag), then we can remove this assignment.

6.5 Verify Security Property with Z3

We run the Boogie application on our generated Boogie source to verify that our security property holds. The Boogie is compiled to a SMT equation which Z3 can then reason about and solve.

7 Property Details

There are several properties which are useful to prove a hypervisor correct. The main property this paper is con-
function ITE<(b : bool, x : a, y : a) returns (a);

axiom (forall <a>b : bool, x : a, y : a :: {ITE(b,x,y)} (b ==> ITE(b,x,y) == x) & (b !== ITE(b,x,y) == y));

function align(addr : bv32) : bv32;

axiom (forall addr : bv32 :: {align(addr) : bv32} align(addr) == addr[32:2] ++ 0bv2);


axiom (forall mem:[bv32]bv32,addr:bv32,value:bv8 :: {STORE_LIT8(mem,addr,value) : [bv32]bv32}

STORE_LIT8(mem,addr,value) ==
ITE(addr[2:0] == 0bv2,mem[align(addr) := mem[align(addr)][32:8]++value],
ITE(addr[2:0] == 1bv2,mem[align(addr) := mem[align(addr)][32:16]++value++mem[align(addr)][8:0]],
ITE(addr[2:0] == 2bv2,mem[align(addr) := mem[align(addr)][32:24]++value++mem[align(addr)][16:0]],
ITE(addr[2:0] == 3bv2,mem[align(addr) := value++mem[align(addr)][24:0]],mem)))));

Figure 4: Boogie model for storing 8-bit integers to memory

ensures (forall i : bv32 ::
((GE_32_32(i, 2097152bv32) &&
LT_32_32(i, 2103848bv32)) ||
(GE_32_32(i, 3221225472bv32) &&
LT_32_32(i, 3221264363bv32)))
==> old(mem[i]) == mem[i]);

Figure 6: This Boogie property verifies that the code pages remain unchanged. The code pages are between addresses 2097152 and 2103848, and 3221225472 and 3221264363.

cerned with is proving that the code page for the hypervisor is not modified. This is a necessary requirement to prove further properties, because without it, we would be required to reason about all possible instruction sequences which could exist. We prove this in three pieces. In the current work, we prove that the intercept handler for the hypervisor does not modify the hypervisor’s own code pages. This should not happen if the hypervisor was carefully programmed. We believe the trade off of only being able to verify hypervisors that do not self modify their code is reasonable.

In future work, we hope to show that the guest can not directly write to the hypervisor’s code pages during its execution. Allowing this would be even more unusual than the hypervisor changes its own code. This protection would be implemented by the hypervisor not mapping the guest into the hypervisors code pages. Finally, we must simply assume the hardware does not directly write to the hosts memory.

The Boogie definition of our property is shown in Figure 6. It states that the ELF init and text regions of memory should be the same before and after execution of the intercept handler.

In the future, we would like to add more complicated properties which prove that guests running on the hypervisor are isolated from each other. This would require we show that there are no information channels between guests, both direct (e.g., guests can not write to each other’s memory) and indirect (e.g., the hypervisor does not copy data from one guest to the other). We will also prove correctness with devices which have Direct Memory Access (DMA) capabilities by verifying the IOMMU is properly enabled.

8 Test Cases

Since we do not have the ground truth for whether or not NOV A actually does protect its own code pages (we would expect it would, but we can’t be certain), we instead demonstrate the ability to detect known bugs and detect bugs we injected into NOV A. We are not aware of any bugs in any NOV A version that causes modification of the code pages; we both examined the development repository for previous and discussed with the NOV A developers.

With no existing bug for our code page property, we added a second property and ported a bug from a different hypervisor to NOV A. MinVisor [5] relied on preventing its guests from entering 64-bit mode yet did not prevent the IA32_EFER.LME bit that enables long mode from being set in the VMCS guest save area. We created a property that tested if the save state of the guest set long mode. We did a very crude port of the MinVisor bug to NOV A by having the interrupt handler set that bit in the restore state (as the management guest did) and then not clear it properly (as the bug in MinVisor also failed to do). The port and property are shown in Figure 7. Our no-long-mode property passes when the hypervisor ensures long mode is disabled. When the hypervisor’s clearing of the long mode bit is commented out, the no-long-mode property fails.

For our property of the hypervisor protecting its code pages, we injected artificial bugs into NOV A that would violate this. The first bug simply wrote to the address of the intercept handler function. The second bug, given in Figure 8, read the VM exit reason from the VMCS, masked the low bit, and used that as an index to a two
Port of bug:

// Bad manager sets guest start state
// with insecure enable of long mode.
asm volatile(
  "movl $0x00002806,%ebx\n"
  "vmread %ebx,%eax\n"
  "orl $0x00000100, %eax\n"
  "vmwrite %eax,%ebx" );

/* // Hypervisor clears it, or forgets..
asm volatile(
  "movl $0x00002806,%ebx\n"
  "vmread %ebx,%eax\n"
  "andl $~0x00000100, %eax\n"
  "vmwrite %eax,%ebx" ); */

Security property:
ensures vmcs[10246bv32][9:8] == 0bv1;

Figure 7: MinVisor bug ported to NOVA and property.

byte array just below the code segment, adding one to the
index produced a buffer overflow into the code pages. We
compiled two version of Nova with and out these
bugs introduced, which we believe are reasonable bugs
to occur in a hypervisor. When present, each of these
was detected by our tool.

The longest trace we generate has 81 conditional
branches. While it may seem that this implies there are
at least 2^{81}
possible traces through the binary, this is not
the case. Because conditional jumps do not jump outside
of the function they begin within, it is probable that in
reality there are only tens of thousands of possible traces
through the binary. (That is, there may be 2^{10}
terms through eight different functions; so to cover all paths
one would only need to run 8 * 2^{10}
different traces.)

9 Results for Unknown Bug Finding

We ran our tool on 1000 random traces obtained from
our modified Nova binary. Random traces were obtained
in the method described in Section 6.1.

Of the 1000 random traces, when we included path
constraints all 1000 properly verified in under three sec-
onds, see figure 9 for a plot of verification time versus
number of writes. Runtime is correlated with memory
reads but not with the number of instructions of the trace.
Each trace took between 0.5 and 1.0 seconds to generate.

In order the demonstrate the efficacy of path con-
straints, we removed path constraints and reran our tests
on 100 random paths. 29 of the 100 paths timed out, tak-
in more than five minutes. The next 66 took under 2
seconds, and the remaining five took between 2 seconds
and 5 minutes..

10 Discussion

A result of our work is clarification of what programming
constructs make verification difficult in hypervisors. For-
tunately, for most of these constructs there is an alternate
which is both equally powerful and simpler to prove cor-
rect.

First, reasoning about potentially infinite linked lists
is a difficult task. Placing a finite bound on the length,
when possible, allows verification to proceed without
significant trouble as the loop can be unrolled to the max-
umum allowed depth.

Indirect jumps are difficult to reason about because
they can transfer the instruction pointer to any address
in memory. This makes it extremely difficult to reason
about, because jumps may occur to the middle of instruc-
tions and result in different decodings than the intended
ones. We found two ways in which indirect jumps are
used, and both of these can be replaced with other constructs, albeit with a performance penalty.

The first usage of indirect jumps is in switch statements. When constructing long switch statements, the compiler can insert an indirect jump to go to the correct case without testing each case in turn. This can be avoided by asking the compiler to not use an indirect jump at the switch, and instead compile it as nested conditional jumps.

The second usage of indirect jumps we found was as function pointers. This is to be expected: the purpose of function pointers is to allow calling of any function. However, importantly, function pointers in C can only point to the beginning of a function. Even if the analysis could only assume the result of jumping to the function pointer would bring the instruction pointer to one of the defined functions, this would significantly simplify analysis, as there are now many fewer possibly entry points.

To further simplify analysis, we found it is easy to replace all instances of indirectly jumping to a function pointer with a switch on the value of the function pointer, and then jumping to the value after that. This allows the analysis tool to reason about only the possible jump targets that will actually be used, instead of all possible jump targets. In Boogie specifically, this helps us as jump targets must be declared ahead of time, and so this does not require recompilation, as some other automated methods might (e.g., the counterexample-driven enlargement of possible jump targets).

11 Future Work

Our tool currently relies on traces to attempt to identify bugs. Due to the large number of possible paths, in the future we will need to consider multiple traces at once. In order to reduce the complexity of the SAT formula, we plan to take a hybrid approach: we will, for the first 5 to 10 branches, we explore all paths, after which we generate the full program possible from that location.

We will need to handle indirect jumps to totally verify NOVA. We also have a plan for this, whereby each indirect jump will be converted to a switch of a finite number of direct jumps. We will identify all possible indirect jump locations through an iterative counter-example driven procedure. To discover what addresses are reachable, we initially assume that none are. At the indirect jump location in the Boogie code, we will assert that the actual jump target is a member of the set of possible addresses, initially empty. If it is, then we jump to that location. When the jump is to a new location not in the set, the assert fails and we obtain a counter-example. We then re-generate the boogie with this location added to the set. This procedure is sound, and assuming there are only a finite number of addresses, this process will eventually terminate.

It will at some point become necessary to model the initialization code. This will introduce several challenges, primarily due to the length of the initialization code. There is an order of magnitude more initialization code than intercept handler code.

In particular, we expect that we will need to use loop invariants for this analysis. While all loops in the initialization are bounded, they are bounded by very large numbers (e.g., a million). While in theory it is possible to unroll a loop of this size, it is practically infeasible.

There are many other security properties we want to add in the future. Properties about the state of the machine, such as a guest being limited to a resource subset, should be readily expressed in Boogie and thus testable. Properties about guests resources being disjoint would also seem readily testable. Properties about not leaking information from one guest to the other through the hypervisor may be harder to express.

12 Conclusion

In this paper we demonstrated the feasibility of verifying the correctness of the NOVA hypervisor with limited manual analysis. We constructed a model of the BAP IR and used the toil utility, to which we added system-mode semantics, to verify that the hypervisor does not modify its own code pages.

When we injected three bugs into the NOVA binary, our tool was able to find these bugs. Two of these bugs related to the example property we used throughout, that the code page is not modified. We ported one bug which existed in a real hypervisor to NOVA, added a property which checked the error that existed, and our tool successfully found the bug.

Finally, we extracted 1000 random traces out of the intercept handler from NOVA and verified that all of these traces preserve our security property. We discuss our results and identify features of hypervisor design which complicate automated verification, and suggest alternate features which are simpler to verify.

References


